# Project Integration Architecture: Distributed Lock Management, Deadlock Detection, and Set Iteration

William Henry Jones

April 26, 1999

## 1 Abstract

The migration of the Project Integration Architecture (PIA) to the distributed object environment of the Common Object Request Broker Architecture (CORBA) brings with it the nearly unavoidable requirements of multi-accessor, asynchronous operations. In order to maintain the integrity of data structures in such an environment, it is necessary to provide a locking mechanism capable of protecting the complex operations typical of the PIA architecture. This paper reports on the implementation of a locking mechanism to treat that need, and upon the ancillary features necessary to make that mechanism work.

## 2 Introduction

The Project Integration Architecture (PIA) is an object-oriented architecture within which practically any engineering application may be wrapped. Information in this architecture is provided not only through the isolated objects which it presents, but also through the structural relationships of those objects to one another. For instance, while engineering data is maintained in a configuration object in a conceptually-flat, balanced, binary tree, the logical organization of that data into structural units intuitive to the application user is revealed by an accompanying, n-ary tree of identification objects. The revelation of information through structure brings with it the consquence that many, if not all, PIA transactional operations involve sets of objects rather than single objects.

In migrating the Project Integration Architecture to the Common Object Request Broker Architecutre (CORBA) environment of distributed objects, the complexities of multi-accessor operations are brought into the design mix. While one might wish to serve the objects of a particular PIA application instance to a single client, nothing in the basic CORBA specification allows for such a restriction. Thus, it is appropriate, if perhaps not explicitly necessary, to provide for the locking of such transactional object sets.

A simple, mutual-exclusion semaphore locking capability, as is commonly provided in many software products and environments, is not appropriate to the task of locking multiple objects. Such single locks control single entities. Thus, for a single semaphore to be effective, it would have to be understood as protecting the single set of objects to manipulated in a transaction; however, since that object set is dynamically determined at the time the transaction is proposed, such a single lock can not be pre-established and recognized by the multiple clients which might interfer in such a transaction. Further, since another client's inter-

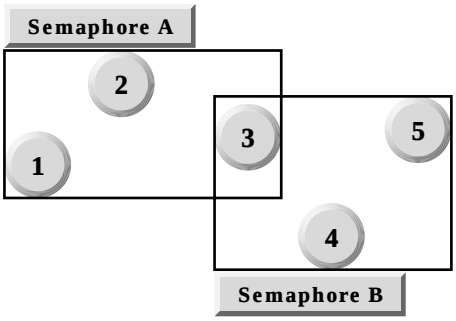


Figure 1: Which Semaphore for Intersecting Sets?

ference might involve a distinct object set not identical to the first set, yet nevertheless intersecting that first locked set, the efficacy of the single, semaphore-like lock approach is clearly lacking. This is illustrated in Figure 1.

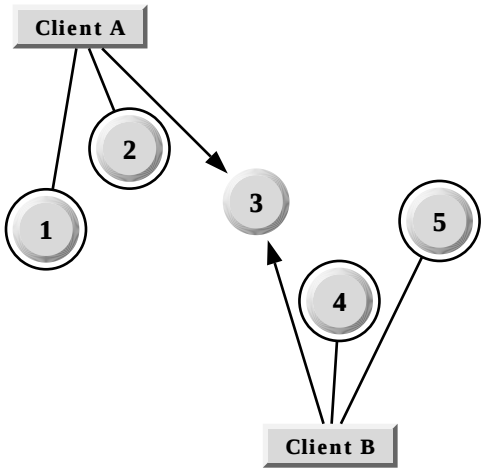


Figure 2: Client Transactions Compete for Distributed Locks

To control dynamically selected sets of objects, it is necessary to provide a matching set of locks, each lock dedicated to a particular object. By having a distinct, identifiable lock associated with each indivisible object, competing transactions may contend with each other for control of the individual objects necessary to make the transaction go forward, as is depicted in Figure 2. Additionally, it is desirable for such locks to provide not merely a yes/no response to an across-the-board usage request, but a graded set of access levels to the controlled object so that transactions with compatible needs (in particular, read access) may progress together while assuring that transactions with conflicting requirements (in particular, write or delete access) are excluded.

Another desirable aspect of a lock mechanism is that it be distributed, even as was implicitly suggested in Figure 2. This is, perhaps, more clearly seen when the statement is examined from the contrapositive view: a centralized lock management system is very undesirable. In a centralized lock system, the rate at which transactions can proceed very rapidly becomes determined by the rate at which the centralized lock system can process lock operations. By distributing lock operations, the operational resources of the transaction may be brought to bear upon the lock operation, too. Thus, as transactional resources grow (through multiply-threaded servers and multi-server environments), lock processing resouces grow proportionately.

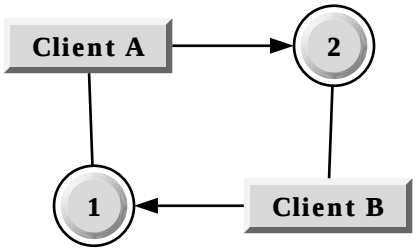


Figure 3: A Simple Deadlock Condition

With the introduction of multiple-lock environments, the possibility of deadlock, an irresolvable conflict in the holding and requesting of locks between two or more clients, is introduced as well. In its simplest form (depicted in Figure 3), a deadlock occurs when client A holds a lock on object 1 and desires a lock on

object 2 while client B holds a lock on object 2 and desires of lock on item 1. As the example suggests, the detection of deadlocks is relatively straightforward; however, it, in turn, brings with it the problem of iteration upon a dynamically changing set, in this case, the set of lock holders. Furthering the example, there may be a client C holding a lock on object 2 with no designs on object 1. The transaction of client C may run to completion and release the lock on object 2 while client A is performing its evaluation of the deadlock condition.

Reviewing all of the above, distributed lock management brings with it a series of interesting problems. Each are amenable to solution and, while each solution is no particular act of genius, the effort as a whole may be instructive as to the issues that must be confronted in providing meaningful locks in a distributed, structural, object environment.

# 3 The Solution

The solution of the lock management problem posed above involves three interfaces in the CORBA environment: a lock, a lock context, and a positional iterator. As shown in Figure 4, an instance of the lock interface, GLock, is attached to a lockable interface instance. A lock context associated with a client is supplied to the lockable instance to provide a context within which a lock may be held. A positional iterator instance (not shown in the figure) is created internally by the lock context and is initialized and maintained by the lock in the event that an evaluation of a potential deadlock condition must be made.

## 3.1 The Lock and Lock Context

The lock interface, GLock in this implementation, provides, as would be expected, the basic locking function. That is, it provides a decision to a requester whether or not, at the present time, a requested form of access can be granted.

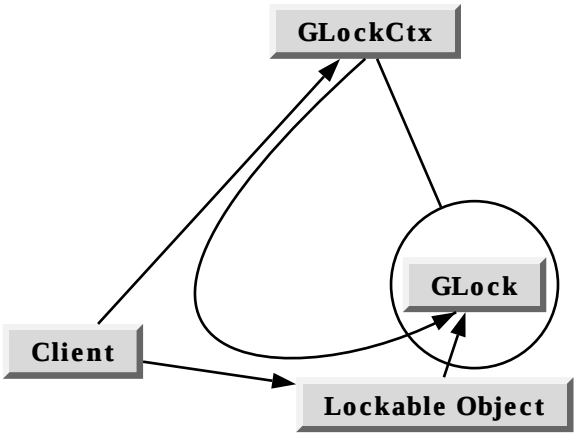


Figure 4: Relationship of Principal Distributed Lock Components

This decision form of lock management is in opposition to the alternative blocking form in which a process requesting a resource lock is suspended until the lock is granted or a deadlock exception (of whatever form) is thrown. Here, the GLock service responds to a lock request not by blocking the requester until the lock can be granted, but by simply issuing a yes-or-no decision. The decision form was necessary because of the lack of any generic, cross-platform capacity for suspending an executing thread and queuing it on some subsequent lock event.

The GLock interface, as it is currently implemented, recognizes five locking levels: Release, Reference, Read, Write, and Delete.

1. The holding of a Release lock grants, paradoxically, a complete disassociation of the requester from the lockable entity. It is introduced into the lock design for the clarity of the implementation. By its nature, a Release lock may be obtained by any requester at any time. Thus, a new requester may be immediately granted a Release lock and, as a result, the lock granting process

(and lock releasing process) may be transformed in all cases to a lock conversion process.

2. The holding of a Reference lock grants to the requester the right to expect the lockable entity to continue to exist. The locked entity may be read, written, and otherwise change its state as a result of operations carried out by other clients obtaining appropriate locks, but it may not cease to exist under the operations appropriate to the granting of a Delete lock. Multiple requesters may hold this lock simultaneously and this lock may be held in the presence of other Release, Read, and Write locks.

3. The holding of a Read lock grants the right to the requester to obtain, but not change, the state of the lockable entity. Multiple requesters may hold this lock simultaneously and this lock may be held in the presense of one or more Reference locks; however, this lock may not be held in the presence of any Write or Delete lock.

4. The holding of a Write lock grants the right to the requester to obtain, modify, and or set the state of the lockable entity. At most one requester may hold a Write lock at a given time and the lock may only be held in the absense of any and all other Read and Delete locks.

5. The holding of a Delete lock grants the right to the requester to remove from operation and discard (in whatever sense) the lockable entity. Subsequent to the operations permitted by a Delete lock, the expectation is that the lockable entity will no longer exist in any meaningful, operational sense. At most one requester may hold a Delete lock at a given time and the lock may only be held in the absense of any and all other Reference, Read, and Write locks.

Largely because of the Release lock device, the lock request process can be implemented, in its essense, as a simple finite state machine based upon the following state variables: the current lock granted to the requester, the lock requested by the requester, and the current lock state of the GLock instance. With the addition of a few amenities such as the use of a mutual exclusion semaphore to protect the internals of the particular GLock instance, the unconditional granting of a Release locks to requesters with no current lock, and the discarding of any granted Release locks at the conclusion of the lock process, the basic function of the GLock interface is complete.

A small further adjustment exists in the basic lock process. Because of the multiple-reader, single-writer protocol specified above, it is possible for multiple, sequential Read locks to block a Write lock for an indefinite period of time, even though no fundamental write inhibition exists. To adjust for this problem, the implemented GLock interface will suspend the granting of Read locks for a short period of time after the refusing of a Write lock request. This is done in the expectation that no requester will simply make one request and give up. Instead, it is expected that the Write lock request will be repeated shortly during the period in which Read locks are being declined and that, during that period, existing Read locks will be completed and released, allowing the Write lock request to be granted. It is further supposed that the write operation will complete shortly and that refused Read locks will be granted on subsequent request.

It should be noted explicitly for the purposes of later discussion that the lock instance maintains a map of lock contexts holding locks on it. This map includes the level of lock each such context holds.

The lock context interface, GLockCtx, provides the operating context in which the set of locks necessary for a single, logical transaction is held. Typically, a single lock context is utilized by a client to hold the locks of that client. In terms of basic function, the lock context is

not particularly complicated.

The lock context handles the mechanics of requesting a lock on a particular GLock lock instance, implementing the retry protocol mentioned above when locks are refused. Thus, when the lock context reports that a lock has been refused, that lock has already been requested and denied several times.

Also the lock context provides a programmatically useful lock-nesting concept. A particular operation may request a lock nest and obtain multiple locks within it. When the operation is complete, it may rely on the unnesting operation of the lock context to release those locks to their previous state. Through a programmatic slight-of-hand, this nesting capability may be used to assure the release of obtained locks even when exceptions are thrown past the operational scope in which the locks were obtained.

## 3.2 Deadlock Detection

When a lock context repeatedly is refused a lock it is requesting, it is of interest to determine whether such a refusal represents a deadlock condition (as depicted in Figure 3 on page 2), or whether it is the result of some more indefinite (and probably irresolvable) condition. This task falls (in this implementation) to the lock context interface, though the facilities of the lock interface provide key information in this operation, and it is in that lock context that a deadlock declaration is made.

It is true that, if client A is deadlocked because of client B, client B is then also deadlocked because of client A; however, the approach implemented here leaves it to each client's lock context to detect that reciprocal truth for itself. Thus, it may be that if client A detects and resolves the deadlock with client B, client B may not ever necessarily identify that the reciprocal deadlock condition existed.

As mentioned previously, the simplest form of deadlock occurs when client A holds a lock on object 1 and requests a lock on object 2 while client B holds a lock on object 2 while requesting a lock on object 1. The implemented lock system refines this basic example to include the concept of conflicting locks based upon the multiple-reader/single-writer protocol the system implements. Clearly, if clients A and B are holding and requesting Read locks, no deadlock condition exists.

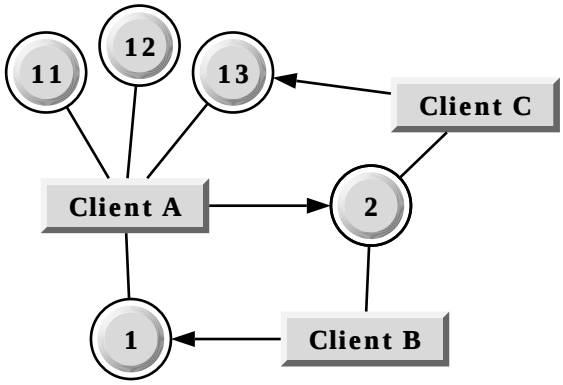


Figure 5: Multiple Direct Deadlock Conditions

The next step in widening the deadlock detection process is to recognize, as depicted in Figure 5, that there may be multiple holders of conflicting locks on the object upon which a particular client desires a lock. That is, client A holding locks on objects 1, 11, 12, and 13 and desiring a lock on object 2 may find that both clients B and C hold conflicting locks on object 2 and either one of them may cause a deadlock by requesting a lock on any of the objects locked by client A. Additionally, it is important to note that the nature of the conflicts between client A and client B and between client A and client C need not be the same.

While all of this is already an interesting algorithmic exercise, it is still not a sufficient definition of a deadlock condition. As shown in Figure 6, deadlock can result from a chain of locks held and requested. That is, client A holds a lock on object 1 and requests a lock on object 2 while client B holds a conflicting lock on object

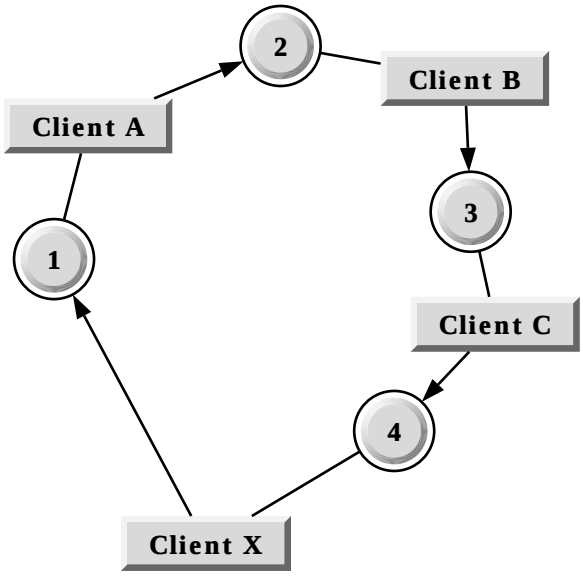


Figure 6: A Chain of Deadlock Conditions

2 and requests a lock on object 3. Meanwhile client C holds a conflicting lock on object 3 and requests a lock on object 4, and so on, until, at last, some client X holds a conflicting lock on an object in the chain and requests a lock on object 1, which cannot be obtained because client A holds the orginal conflicting lock on object 1.

In quasi-technical terms, let a directed graph be formed in which the initial node is the lock context performing the deadlock analysis and, for each such node of the graph, the immediate successor nodes are those nodes holding a conflicting lock on the lock instance of which the subject node is requesting a lock. A deadlock then exists if that graph proves to be cyclic at the initial node.

As noted previously, it is to facilitate this computation that lock instances retain a map of lock contexts holding locks on the presenting lock instance and record the kind of lock held by each such lock context. This is exactly the information required for the deadlock computation. Further, lock contexts are sorted by lock level so that the set of lock contexts holding conflicting locks with a specified lock level may be quickly identified.

The resolution of the deadlock condition is not a particular interest in this paper. Currently, the expectation is that deadlocks will be resolved by releasing and re-obtaining all of a client's locks. The overall project from which this work is reported has not reached the point at which this issue has been decided and, indeed, it may be that multiple resolution strategies are possible.

## 3.3 Set Iteration

The implementation of the deadlock algorithm is, itself, reasonably straightforward; however, one issue does come up: the set of lock holders is, itself, not constant with time. As a particular client makes its way through the calculation, other clients may either release or obtain locks relevant or otherwise. A significant element of the implementation is, thus, not simply implementing the algorithm, but making that algorithm tolerant of the fact that the problem may be changing as it is computed. In particular, another client may already have identified its reciprocal deadlock condition and be in the act of releasing locks as its resolution method.

This is, in fact, a particular case of a general problem in multi-accessor environments: iterations upon a set must be tolerant of the fact that the set being iterated upon may change during the course of iteration. Because of this dynamicism, direct iteration upon structures such as linked lists, directed graphs, and the like is inadvisable. A reference held by a client to the next element of an iteration may become invalid due to the operation of another client upon the set. For example in a linked list, should a linked element regarded as the next element of traversal by one client be removed from the list by another client, the first client will have the nasty problem not only of having a next reference to an element that no longer

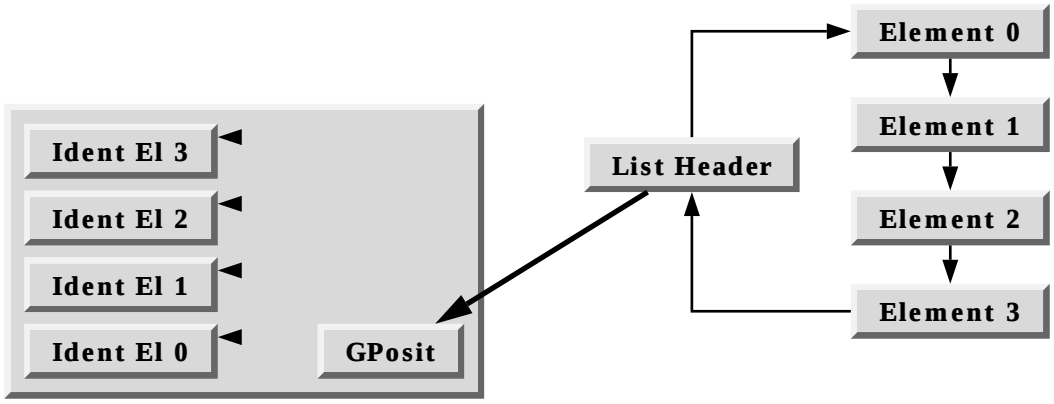


Figure 7: An Initialized GPosit Iterator

exists, but also of having to re-establish its current operating position in that list.

A solution to this problem is, of course, possible by the simple expedient of locking up the entire structure on which the iteration is to occur, perhaps through some protocol of obtaining a Read lock on a header or controller element. The difficulty with this solution is exactly what it does: it locks up the entire structure for the duration of the iteration. If one assumes that iterative processes will be inherently fast, that multiple iterators on a given set will be generally rare, or that such iterations will be generally disruptive of simultaneous operations anyway, then such a policy is, perhaps, not a bad choice. On the other hand, if any or all of the opposites are generally true, then locking up entire iterative structures for iterative traversals very quickly leads to one iterator blocking all others for the duration of its operation.

To solve this problem, something called a positional iterator has been devised in the form of the GPosit interface. A private instance of this interface is obtained for each client iteration. The GPosit instance is initialized with the identifications of each element of the iterative set, as depicted in Figure 7. This initialization does, indeed, employ the solution above of locking the entire iterative set for the duration of that initialization; however, it is hoped (if not proven by practical experience) that the traversal for the purpose of identification only will be reliably faster than traversal for the real purpose of iterative operation, whatever that real computational purpose might be. For the deadlock operation, this means that the lock instance will (quickly) scan all of the lock contexts holding a conflicting lock on that lock instance and load their identifications into the GPosit iterator supplied by the client's lock context deadlock algorithm. Because of the internal arrangements made in the lock interface, this process proceeds with considerable efficiency.

The next key element of this set iteration scheme is that, as a final step of iterator initialization, the iterator instance is made a primative successor of the interface instance controlling additions to and removals from the iterative set. In the case of the deadlock algorithm, this means that each GLock instance initializing an iterator makes that iterator a successor of itself and, in this application, also notes the lock level with which the iterator was concerned. In other iterative sets, this requirement to be a successor of a controlling instance does place a restriction upon such traversable struc-

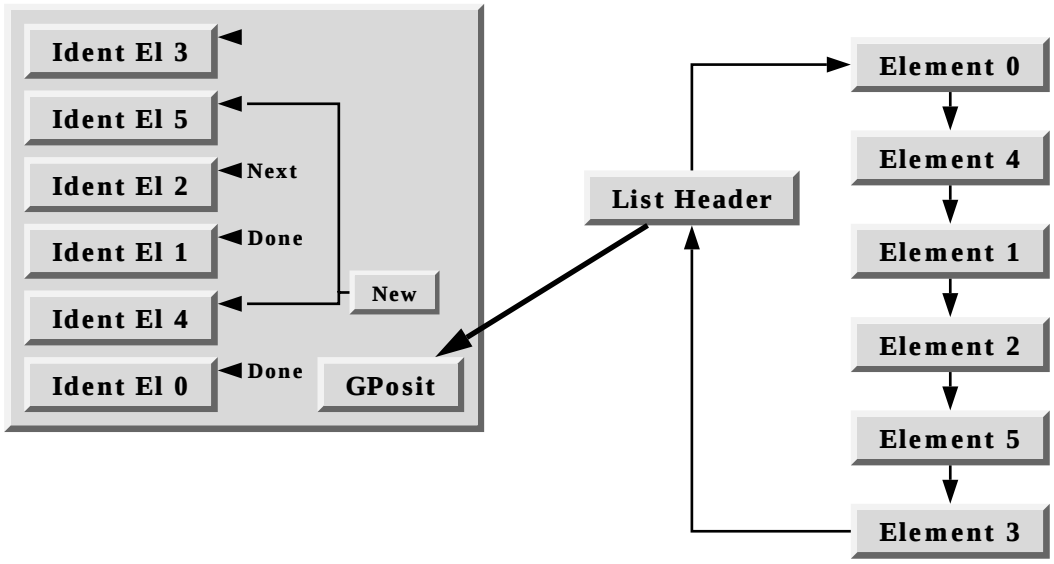


Figure 8: Effect of Set Additions on an Operating Iterator

tures: some single instance must be that controlling, cognizant point. Thus, a linked list cannot be regarded as a perfectly circular list in which any element may be momentarily regarded as the head. Instead, one element must act as the head in all cases so that a central point of control for iterative operations can be maintained.

Having received an iterator, a client may then obtain element identifications from that iterator at its leisure, performing such protracted operations as may be its wont. Provision is made for forward and backward traversals of the set, the identification of set elements in various ways, and for definitive detection of the end-of-set condition.

When the actual iterative set changes, it is the responsiblity of the central point of control to locate each GPosit iterative set successor and notify it of the change. In the case of the deadlock operation, this means that the granting or releasing of a conflicting lock results in the notification of the positional iterator of that change in lock status. (Note that this notification process is sensitive to the lock level associated with each iteration and results in actual notification of the iterator only when that lock event is of interest to that deadlock operation.)

By encapsulating the iterative identification operation in a separate interface instance, a key difference in the treatment of iterative set events is possible. The actual set, the linked list structure or the map of contexts holding locks, may be appropriately changed in response to the event, just as it should be. Meanwhile, the notified GPosit iterator instance makes crucially different adjustments. In the event of addition to the set as shown in Figure 8 (which continues the example of Figure 7 on page 7), not only are the identifications of new elements added at the appropriate point, but the identification of each new element is also retained for special consideration at the time of the next iterative step by the client. But the most important difference is upon a removal event as shown in the example continued by Figure 9: the identification of the removed element is not actually itself removed, but only

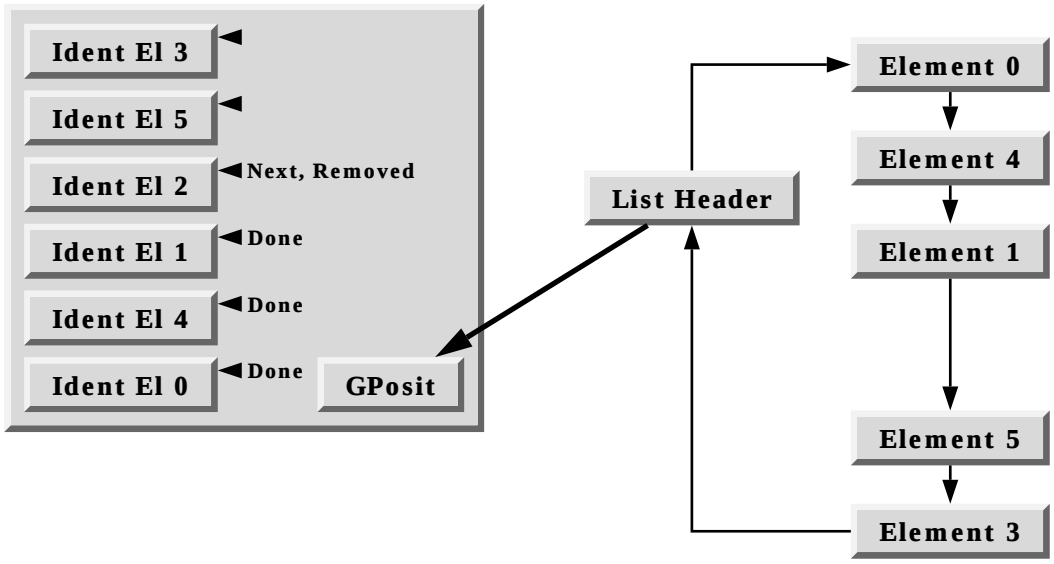


Figure 9: Effect of a Removal from a Set on an Operating Iterator

marked as having been removed. Thus, at the next iterative step, should that removed element also be the next element of the iteration, the position in the set is not lost. It is only necessary for the iteration to recognize the removed status of the next element identification and step over it before reporting the (new) next position to the client.

The treatment of the element addition event by the positional iterator brings on new possibilities. Generally, set iteration is considered an ordered process, proceeding for example forward through a linked list; however, with the positional iterator it is possible to consider the presense of newly added set elements at the time of the next iterative step. If such elements are still further on from the present position of the iteration (as identification element 5 is in Figure 8 on page 8), then they may be simply left to consideration in their proper order; however, should the iteration already have passed the position of one or more newly added elements (as is the case for identification element 4 in Figure 8), it is possible (optionally) to back the iteration up to the point of the earliest such added element. Since the iterative set interface keeps track of both elements done and elements removed, such backing up does not repeat iterative steps, but merely makes the iteration unpredictably non-monotonic in its nature.

In application to the case of deadlock detection, a newly granted lock introducing a new, alternate deadlock condition can be detected dynamically as it occurs. Alternatively, should a reciprocal deadlock condition be preemptively cured by the releasing of the locks held by another client, the release of those locks will also be dynamically noted and avoid the finding of a deadlock condition that, in fact, no longer exists. The key point, though, is that while a particular client context is involved in the potentially lengthy process of deadlock assessment, lock operations for locks involved in that issue can continue. Thus, other clients who in fact come into no deadlocked contention with the assessing client may continue lock transactions and perform useful work.

# 4 Additional Commentary

When the iterative set is internally held, as it is in the case of the GLock interface, some solution in the manner of the positional iterator is mandatory to expose that which is otherwise concealed; however, in the case of exposed iterative structure, as the linked list used in the examples above, such a solution is not required. Direct iteration of such exposed structures is clearly possible. In the case of a linked list, the simple expedient of maintaining a Read lock on the current iteration item will assure that some other accessor does not succeed in removing that item from the list.

Despite the fact that the positional iterator is not strictly necessary in the case of exposed structures, the current expectation is that it will form the basis of the standard iterative mechanism in the CORBA-served PIA migration. The following reasons are put forward in support of this outlook.

1. The use of the positional iterator interface will bring, *ipso facto*, unity to the iterative form. In so doing, coding will be more predictable and less sensitive to the structural form supporting the operation. Later changes in structural form due to software revisions and the like will have less impact.

2. The introduction of derived positional iterator interface forms provides the opportunity to add further internal iterative context for those structural forms for which it is necessary without the necessity of disrupting the basic iterative coding form established by the positional iterator interface. Such derived forms might, indeed, provide direct iteration upon a structure without altering the basic coding form.

3. The ability of the positional iterator to back up to newly added set elements is a useful feature not available to direct iterations. Providing an iterative set event notification to direct iteration code would be an extremely complicated task.

# 5 Concluding Remarks

A solution to the problem of distributed lock management, deadlock detection, and the iteration on dynamic sets needed to solve the deadlock detection problem has been presented. The presented solution is neither perfect nor the only solution possible; however, the work presented is presented as neither of those, but only as a workable solution to a practical problem. It must be remembered that in any multi-accessor, asynchronous operating environment, solutions always represent an engineering tradeoff between flexibility and perfection.